# Глава 9. Архитектура операционных систем

Как комплекс системных управляющих и обрабатывающих программ (см. главу 1), операционная система представляет собой очень сложный конгломерат взаимосвязанных программных модулей и структур данных, которые должны обеспечивать надежное и эффективное выполнение вычислений. Большинство потенциальных возможностей операционной системы, ее технические и потребительские параметры — все это во многом определяется архитектурой системы — ее структурой и основными принципами построения.

В главе 1 мы упомянули несколько наиболее распространенных классификаций. Очевидно, что системы, ориентированные на диалог, должны иметь иные стратегию обслуживания и дисциплину диспетчеризации, чем системы пакетной обработки. Диалоговое взаимодействие предполагает реализацию развитой интерфейсной подсистемы, обеспечивающей взаимодействие пользователя с компьютером. Это отличие сказывается и на особенностях построения систем. Очевидно, что для диалоговых операционных систем необходимо предусмотреть множество механизмов, которые позволят пользователям эффективно управлять своими вычислениями.

Аналогично, и системы, реализующие мультизадачный режим работы, отличаются по своему строению от однозадачных систем. Если система допускает работу нескольких пользователей, то желательно иметь достаточно развитую подсистему информационной безопасности. А это, в свою очередь, налагает определенные требования и на идеологию построения операционной системы, и на выбор конкретных механизмов, помогающих реализовать защиту информационных ресурсов и ввести ограничения на доступ к другим видам ресурсов. Поскольку операционные системы помимо функций организации вычислений и организации интерфейса пользователя предоставляют интерфейсы для взаимодействия программ с операционной системой, мы в этой главе рассмотрим и интерфейсы прикладного программирования.

# Основные принципы построения операционных систем

Среди множества принципов построения операционных систем перечислим несколько наиболее важных: принцип модульности, принцип виртуализации, принципы мобильности (переносимости) и совместимости, принцип открытости, принцип генерации операционной системы из программных компонентов и некоторые другие.

## Принцип модульности

Операционная система строится из множества программных модулей. Под *модулем* в общем случае понимают функционально законченный элемент системы, выполненный в соответствии с принятыми межмодульными интерфейсами. По своему определению модуль предполагает легкий способ его замены другим при наличии заданных интерфейсов. Способы обособления составных частей операционной системы в отдельные модули могут быть существенно разными, но чаще всего разделение происходит именно по функциональному признаку. В значительной степени разделение системы на модули определяется используемым методом проектирования системы (снизу вверх или наоборот).

Особо важное значение при построении операционных систем имеют *привилегированные, повторно входимые* и *реентерабельные* модули, ибо они позволяют более эффективно использовать ресурсы вычислительной системы. Как мы уже знаем (см. главу 1), свойство реентерабельности может быть достигнуто различными способами, но чаще всего используются механизмы динамического выделения памяти под переменные для нового вычислительного процесса (задачи). В некоторых системах реентерабельность программы получают автоматически. Этого можно достичь благодаря неизменяемости кодовых частей программ при исполнении, а также автоматическому распределению регистров, автоматическому отделению кодовых частей программ от данных и помещению последних в системную область памяти, которая распределяется по запросам от выполняющихся задач. Естественно, что для этого необходима соответствующая аппаратная поддержка. В других случаях это достигается программистами за счет использования специальных системных модулей.

Принцип модульности отражает технологические и эксплуатационные свойства системы. Наибольший эффект от его использования достижим в случае, когда принцип распространен одновременно на операционную систему, прикладные программы и аппаратуру. Принцип модульности является одним из основных в UNIX-системах.

Во всех операционных системах можно выделить некоторую часть наиболее важных управляющих модулей, которые должны постоянно находиться в оперативной памяти для более скорой реакции системы на возникающие события и более эффективной организации вычислительных процессов. Эти модули вместе с некоторыми системными структурами данных, необходимыми для функционирования операционной системы, образуют так называемое *ядро операционной системы,* так как это действительно ее самая главная, центральная часть, основа системы.

При формировании состава ядра требуется удовлетворить двум противоречивым требованиям. В состав ядра должны войти наиболее часто используемые системные модули. Количество модулей должно быть таким, чтобы объем памяти, занимаемый ядром, был не слишком большим. В его состав, как правило, входят модули по управлению системой прерываний, средства по переводу программ из состояния счета в состояние ожидания, готовности и обратно, средства по распределению основных ресурсов, таких как оперативная память и процессор. В главе 1 мы уже упоминали, что операционные системы могут *быть микроядерными* и *макроядерными (монолитными)*. В микроядерных операционных системах само ядро очень компактно, а остальные модули вызываются из ядра как сервисные. При этом сервисные модули могут размещаться и в оперативной памяти. В противоположность микроядерным в макроядерных операционных системах главная супервизорная часть включает в себя большое количество модулей. Более подробно о микроядерных и макроядерных операционных системах см. далее.

Помимо программных модулей, входящих в состав ядра и постоянно располагающихся в оперативной памяти, может быть много других системных программных модулей, которые получают название транзитных. Транзитные программные модули загружаются в оперативную память только при необходимости и в случае отсутствия свободного пространства могут быть замещены другими транзитными модулями. В качестве синонима термина «транзитный» можно использовать термин «диск-резидентный».

## Принцип особого режима работы

Ядро операционной системы и низкоуровневые драйверы, управляющие работой каналов и устройств ввода-вывода, должны работать в специальном режиме работы процессора. Это необходимо по нескольким причинам. Во-первых, введение специального режима работы процессора, в котором должен исполняться только код операционной системы, позволяет существенно повысить надежность выполнения вычислений. Это касается выполнения как управляющих функций самой операционной системы, так и прикладных задач пользователей. Категорически нельзя допускать, чтобы какая-нибудь прикладная программа могла вмешиваться (преднамеренно или в связи с появлением ошибок вычислений) в вычисления, связанные с супервизорной частью операционной системы. Во-вторых, ряд функций должен выполняться исключительно централизованно, под управлением операционной системы. К этим функциям мы, прежде всего, должны отнести функции, связанные с управлением процессами ввода-вывода данных. Вспомните основные принципы организации ввода-вывода (см. главу 5): *все операции ввода-вывода данных объявляются привилегированными*. Это легче всего сделать, если процессор может работать, как минимум, в двух режимах: привилегированном (режим супервизора) и пользовательском. В первом режиме процессор может выполнять все команды, тогда как в пользовательском набор разрешенных команд ограничен. Естественно, что помимо запрета на выполнение команд ввода-вывода в пользовательском режиме работы процессор не должен позволять обращаться к своим специальным системным регистрам — эти регистры должны быть доступны только

в привилегированном режиме, то есть исключительно супервизорному коду самой операционной системы. Попытка выполнить запрещенную команду или обратиться к запрещенному регистру должна вызывать прерывание (исключение), и центральный процессор должен быть предоставлен супервизорной части операционной системы для управления выполняющимися вычислениями.

Поскольку любая программа требует операций ввода-вывода, прикладные программы для выполнения этих (и некоторых других) операций обращаются к супервизорной части операционной системы (модуль супервизора иногда называют *супервизором задач)* с соответствующим *запросом.* При этом процессор должен переключиться в привилегированный режим работы. Чтобы программы не могли произвольным образом обращаться к супервизорному коду, который работает в привилегированном режиме, им предоставляется возможность обращаться к нему в строгом соответствии с принятыми правилами. Каждый запрос имеет свой идентификатор и должен сопровождаться соответствующим количеством параметров, уточняющих запрашиваемую у операционной системы функцию (операцию). Поэтому супервизор задач при получении запроса сначала его тщательно проверяет. Если запрос корректный и программа имеет право с ним обращаться, то запрос на выполнение операции, как правило, передается соответствующему модулю операционной системы. Множество запросов к операционной системе образует соответствующий системный *интерфейс прикладного программирования* (Application Program Interface, API).

## Принцип виртуализации

В наше время уже не требуется пояснять значение слова «виртуальный», ибо о виртуальных мирах, о виртуальной реальности знают даже дети. Принцип виртуализации нынче используется практически в любой операционной системе. Виртуализация ресурсов позволяет не только организовать разделение тех ресурсов между вычислительными процессами, которые не должны разделяться. Виртуализация позволяет абстрагироваться от конкретных ресурсов, максимально обобщить их свойства и работать с некоторой абстракцией, вобравшей в себя наиболее значимые особенности. Этот принцип позволяет представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов (мониторов) и использовать единую централизованную схему распределения ресурсов.

Следует заметить, что сама операционная система существенно изменяет наши представления о компьютере. Она виртуализирует его, добавляя ему функциональности, удобства управления, предоставляя средства организации параллельных вычислений и т. д. Именно благодаря операционной системе мы воспринимаем компьютер совершенно иначе, чем без нее.

Наиболее законченным и естественным проявлением концепции виртуальности является понятие *виртуальной машины.* По сути, любая операционная система, являясь средством распределения ресурсов и организуя по определенным правилам управление процессами, скрывает от пользователя и его приложений реальные аппаратные и иные ресурсы, заменяя их некоторой абстракцией. В результате

пользователи видят и используют виртуальную машину как некое устройство, способное воспринимать их программы, написанные на определенном языке программирования, выполнять их и выдавать результаты на виртуальные устройства, которые связаны с реально существующими в данной вычислительной системе. При таком языковом представлении пользователя совершенно не интересует реальная конфигурация вычислительной системы, способы эффективного использования ее компонентов и подсистем. Он мыслит и работает с машиной в терминах используемого им языка.

Чаще виртуальная машина, предоставляемая пользователю, воспроизводит архитектуру реальной машины, но архитектурные элементы в таком представлении выступают с новыми или улучшенными характеристиками, часто упрощающими работу с системой. Характеристики могут быть произвольными, но чаще всего пользователи желают иметь собственную «идеализированную» по архитектурным характеристикам машину в следующем составе.

* Единообразная по логике работы память (виртуальная) достаточного для выполнения приложений объема. Организация работы с информацией в такой памяти производится в терминах работы с сегментами данных на уровне выбранного пользователем языка программирования.
* Произвольное количество процессоров (виртуальных), способных работать параллельно и взаимодействовать во время работы. Способы управления процессорами, в том числе синхронизация и информационные взаимодействия, реализованы и доступны пользователям с уровня используемого языка в терминах управления процессами.
* Произвольное количество внешних устройств (виртуальных), способных работать с памятью виртуальной машины параллельно или последовательно, асинхронно или синхронно по отношению к работе того или иного виртуального процессора, которые инициируют работу этих устройств. Информация, передаваемая или хранимая на виртуальных устройствах, не ограничена допустимыми размерами. Доступ к такой информации осуществляется на основе либо последовательного, либо прямого способа доступа в терминах соответствующей системы управления файлами. Предусмотрено расширение информационных структур данных, хранимых на виртуальных устройствах.

Степень приближения к «идеальной» виртуальной машине может быть большей или меньшей в каждом конкретном случае. Чем больше виртуальная машина, реализуемая средствами операционной системы на базе конкретной аппаратуры компьютера, приближена к «идеальной» по характеристикам машине и, следовательно, чем больше ее архитектурно-логические характеристики отличны от реально существующих, тем больше степень ее виртуальности.

Одним из важнейших результатов принципа виртуализации является возможность организации выполнения в операционной системе приложений, разработанных для другой операционной системы, имеющей совсем другой интерфейс прикладного программирования. Другими словами, речь идет об организации нескольких операционных сред, о чем мы уже говорили в главе 1. Реализация этого принципа позволяет операционной системе иметь очень сильное преимущество перед другими

операционными системами, не имеющими такой возможности. Примером реализации принципа виртуализации может служить VDM-машина (Virtual DOS Machine) — защищенная подсистема, предоставляющая полную среду типа MS DOS и консоль для выполнения DOS-приложений. Как правило, параллельно может выполняться практически произвольное число DOS-приложений, каждое в своей VDM-машине. Такие VDM-машины имеются и в операционных системах Windows[1] компании Microsoft, в OS/2, в Linux.

Одним из аспектов общего принципа виртуализации является независимость программ от внешних устройств, хотя иногда эту особенность выделяют особенно и называют принципом. Она заключается в том, что связь программ с конкретными устройствами производится не в процессе создания программы, а в период планирования ее исполнения. В результате перекомпиляция при работе программы с новым устройством, на котором располагаются данные, не требуется. Этот принцип позволяет одинаково осуществлять операции управления внешними устройствами независимо от их конкретных физических характеристик. Например, программе, содержащей операции обработки последовательного набора данных, безразлично, на каком носителе эти данные будут располагаться. Смена носителя и данных, размещаемых на них (при неизменности структурных характеристик данных), не привнесет каких-либо изменений в программу, если в системе реализован принцип независимости программ от внешних устройств. Независимость программ от внешних устройств реализуется в подавляющем большинстве операционных систем общего применения. Ярким примером такого подхода являются операционные системы с общим названием UNIX. Реализована такая независимость и в большинстве современных операционных систем для персональных компьютеров.

Например, в системах Windows все аппаратные ресурсы полностью виртуализированы, и прямой доступ к ним со стороны прикладных (и системных обрабатывающих) программ однозначно запрещен. В системах Windows NT/2000/XP даже были введены понятия *HAL* (Hardware Abstraction Layer — уровень абстрагирования аппаратуры) и *HEL* (Hardware Emulation Layer — уровень эмуляции аппаратуры), и этот шаг очень помогает в реализации идей переносимости (мобильности) операционной системы.

## Принцип мобильности

Мобильность, или переносимость, означает возможность и легкость переноса операционной системы на другую аппаратную платформу. Мобильная операционная система обычно разрабатывается с помощью специального языка высокого уровня, предназначенного для создания системного программного обеспечения. Такой язык помимо поддержки высокоуровневых операторов, типов данных и модульных конструкций должен позволять непосредственно использовать аппаратные возможности и особенности процессора. Кроме этого, такой язык должен быть широко распространенным и реализованным в виде систем программирования,

[1] Не все операционные системы компании Microsoft, в названии которых слово Windows является основным, поддерживают VDM-машины. В частности, такой возможности нет в системе Windows ME.

которые либо уже имеются на целевой платформе, либо позволяют получать программные коды для целевого компьютера. Другими словами, этот язык системного программирования должен быть достаточно распространенным и технологичным. Одним из таких языков является язык C. В последние годы язык C++ также стал использоваться для этих целей, поскольку идеи объектно-ориентированного программирования оказались плодотворными не только для прикладного, но и для системного программирования. Большинство современных операционных систем были созданы именно как объектно-ориентированные.

Обеспечить переносимость операционной системы достаточно сложно. Дело в том, что архитектуры разных процессоров могут очень сильно различаться. У них может быть разное количество рабочих регистров, причем часть регистров может оказаться контекстно-зависимыми, как это имеет место в процессорах с архитектурой ia32. Различия могут быть и в реализации адресации. Более того, для операционной системы важной является не только архитектура центрального процессора, но и архитектура компьютера в целом, ибо важнейшую роль играет подсистема ввода-вывода, а она строится на дополнительных (по отношению к центральному процессору) аппаратных средствах. В таких условиях сделать эффективным код операционной системы при условии создания его на языке типа C/C++ невозможно. Поэтому часть программных модулей, которые более всего зависят от аппаратных особенностей процессора, от типов поддерживаемых данных, способов адресации, системы команд и других важнейших моментов, разрабатывается на языке ассемблера. Очевидно, что модули, написанные на языке ассемблера, при переносе операционной системы на процессор с иной архитектурой должны быть написаны заново. Зато остальная (большая) часть кода операционной системы может быть просто перекомпилирована под целевой процессор. Именно по этому принципу в свое время была создана операционная система UNIX. Относительная легкость переноса этой системы на другие компьютеры позволила сделать ее одной из самых распространенных. Для обеспечения мобильности был даже создан стандарт на интерфейс прикладного программирования, названный POSIX (Portable Operating System Interface for Computer Environments — интерфейс прикладного программирования для переносимых операционных систем).

К сожалению, на самом деле далеко не все операционные системы семейства UNIX допускают относительно простую переносимость созданного для них программного обеспечения, хотя сами они и поддерживают такую переносимость. Основная причина тому — отход от единого стандарта API — POSIX. Очевидно, что платой за универсальность, прежде всего, является потеря производительности при выполнении операций ввода-вывода и вычислений, связанных с этими операциями. Поэтому ряд разработчиков шли и до сих пор идут на отказ от принципа мобильности, поскольку не всегда следование этому принципу экономически оправдано.

Если при разработке операционной системы сразу не следовать принципу мобильности, то в последующем очень трудно обеспечить перенос на другую платформу как самой операционной системы, так и программного обеспечения, созданного для нее. Например, компания IBM потратила долгие годы на перенос своей операционной системы OS/2, созданной для персональных компьютеров с процессора-

ми архитектуры ia32, на платформу PowerPC. Но даже если изначально в спецификации на операционную систему заложить требование легкой переносимости, это не значит, что его в последующем будет просто реализовать. Подтверждением тому является тот же проект OS/2-Windows NT. Как известно, проект Windows NT обеспечивал работу этой операционной системы на процессорах с архитектурой ia32, MIPS, Alpha (DEC), PowerPC. Однако в последующем трудности с реализацией этого принципа привели к тому, что нынешние версии операционных систем класса Windows NT (Windows 2000/XP) уже создаются только для процессоров с архитектурой ia32 и не поддерживают MIPS, Alpha и PowerPC.

## Принцип совместимости

Одним из аспектов совместимости является способность операционной системы выполнять программы, написанные для других систем или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой операционной системе. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего транслятора в составе системного программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того чтобы один компьютер выполнял программы другого (например, программу для персонального компьютера типа IBM PC хочется выполнять на компьютере типа Mac от фирмы Apple), этот компьютер должен работать с машинными командами, которые ему изначально непонятны. Например, процессор типа PowerPC на Mac должен исполнять двоичный код, предназначенный для процессора i80x86. Процессор 80x86 имеет свои собственные дешифратор команд, регистры и внутреннюю архитектуру. Процессор PowerPC имеет другую архитектуру, он не понимает непосредственно двоичный код 80x86, поэтому должен выбрать каждую команду, декодировать ее, чтобы определить, для чего она предназначена, а затем выполнить эквивалентную подпрограмму, написанную для PowerPC. К тому же у PowerPC нет в точности таких же регистров, флагов и внутреннего арифметико-логического устройства, как в 80x86, поэтому он должен эмулировать все эти элементы с использованием своих регистров или памяти. И он должен тщательно воспроизводить результаты каждой команды, что требует специально написанных подпрограмм для PowerPC, гарантирующих, что состояние эмулируемых регистров и флагов после выполнения каждой команды будет в точности таким же, как и на реальном процессоре 80x86. Выходом в таких случаях является использование так называемых прикладных сред, или эмуляторов. Учитывая, что основную часть программы, как правило, составляют *вызовы библио-*

*течных функций,* прикладная среда имитирует библиотечные функции целиком, используя заранее написанную библиотеку функций аналогичного назначения, а остальные команды эмулирует каждую по отдельности.

Одним из средств обеспечения совместимости программных и пользовательских интерфейсов является соответствие стандартам POSIX. Эти стандарты позволяют создавать программы в стиле UNIX, которые впоследствии могут легко переноситься из одной системы в другую.

## Принцип генерируемости

Согласно принципу генерируемости исходное представление центральной системной управляющей части операционной системы (ее ядра и основных компонентов, которые должны постоянно находиться в оперативной памяти) должно обеспечивать возможность настройки, исходя из конкретной конфигурации конкретного вычислительного комплекса и круга решаемых задач. Под генерацией операционной системы понимается ее сборка (компоновка) из отдельных программных модулей. В результате генерации получают скомпонованные двоичные коды операционной системы и построенные системные таблицы, отражающие конкретную конфигурацию компьютера. Эта процедура проводится редко перед достаточно протяженным периодом эксплуатации операционной системы. Процесс генерации осуществляется с помощью специальной программы-генератора и соответствующего входного языка для этой программы, позволяющего описывать программные возможности системы и конфигурацию машины. В результате генерации получается полная версия операционной системы. Сгенерированная версия операционной системы представляет собой совокупность системных наборов модулей и данных.

Упомянутый раньше принцип модульности положительно проявляется при генерации операционной системы. Он существенно упрощает ее настройку на требуемую конфигурацию вычислительной системы. В наши дни при использовании персональных компьютеров с принципом генерируемости операционной системы можно столкнуться разве что при работе с Linux. В этой UNIX-системе имеется возможность не только использовать какое-либо готовое ядро операционной системы, но и самому сгенерировать (скомпилировать) такое ядро, которое будет оптимальным для данного конкретного персонального компьютера и решаемых на нем задач. Кроме генерации ядра в Linux имеется возможность указать и набор подгружаемых драйверов и служб, то есть часть функций может реализовываться модулями, непосредственно входящими в ядро системы, а часть — модулями, имеющими статус подгружаемых, транзитных.

В остальных современных распространенных операционных системах, в том числе и для персональных компьютеров, конфигурирование системы под соответствующий состав оборудования осуществляется на этапе установки, причем в большинстве случаев не представляется возможным серьезно вмешаться в этот процесс. В дальнейшем, при эксплуатации компьютера, можно изменить состав драйверов, служб, отдельных параметров и режимов работы. Как правило, внесение подобных изменений может быть осуществлено посредством редактирования конфигу-

рационного файла или реестра. Например, мы можем отключить ненужное устройство, заменить для какого-нибудь устройства драйвер, отключить или добавить ту или иную службу. Более того, для большей гибкости часто вводится механизм поддержки нескольких конфигураций. Например, такие популярные системы, как Windows 98 и Windows NT/2000/XP, предоставляют возможность создавать до девяти конфигураций. При загрузке операционной системы пользователю предоставляется возможность выбрать одну из имеющихся конфигураций. Таким образом, имея всего одну операционную систему, за счет нескольких различающихся конфигураций пользователь может получить несколько виртуальных систем, различающихся составом установленного (работающего) оборудования, драйверов и служб, и на выбор запускать одну из этих систем.

## Принцип открытости

Открытая операционная система доступна для анализа как пользователям, так и системным специалистам, обслуживающим вычислительную систему. Наращиваемая (модифицируемая, развиваемая) операционная система позволяет не только использовать возможности генерации, но и вводить в ее состав новые модули, совершенствовать существующие и т. д. Другими словами, необходимо, чтобы можно было легко внести дополнения и изменения, если это потребуется, не нарушая целостности системы. Прекрасные возможности для расширения предоставляет подход к структурированию операционной системы по типу клиент-сервер с использованием микроядерной технологии. В соответствии с этим подходом операционная система строится как совокупность привилегированной управляющей программы и набора непривилегированных служб — «серверов». Основная часть операционной системы может оставаться неизменной, в то время как добавляются новые службы или изменяются старые.

Этот принцип иногда трактуют как расширяемость системы.

К открытым операционным системам прежде всего следует отнести UNIX-системы и, естественно, системы Linux.

## Принцип обеспечения безопасности вычислений

Обеспечение безопасности при выполнении вычислений является желаемым свойством для любой многопользовательской системы. Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией многих операционных систем. Для решения этой проблемы чаще всего используется механизм учетных записей. Он предполагает проведение *аутентификации* пользователя при его регистрации на компьютере и последующую *авторизацию,* которая определяет уровень полномочий (прав) пользователя (об аутентификации и авторизации пользователей см. главу 1). Каждая учетная запись может входить в одну или несколько групп. Встроенные группы, как правило, определяют

права пользователей, тогда как создаваемые администратором группы (их называют группами безопасности) используются для определения разрешений в доступе пользователей к тем или иным ресурсам. Имеющиеся учетные записи хранятся в специальной базе данных, которая бывает доступна только для самой системы. Для этого файл базы данных с учетными записями открывается системой в монопольном режиме, и доступ к нему со стороны любого пользователя становится невозможным. Делается это для того, чтобы нельзя было получить базу данных с учетными записями. Если получить файл с учетными записями, то посредством его анализа можно было бы узнать пароль пользователя, по которому осуществляется аутентификация.

Во многих современных операционных системах гарантируется степень безопасности данных, соответствующая уровню C2 в системе стандартов США. Основы стандартов в области безопасности были заложены в документе «Критерии оценки надежных компьютерных систем». Этот документ, изданный в США в 1983 году. Национальным центром компьютерной безопасности (National Computer Security Center), часто называют Оранжевой книгой.

В соответствии с требованиями Оранжевой книги безопасной считается система, которая «посредством специальных механизмов защиты контролирует доступ к информации таким образом, что только имеющие соответствующие полномочия лица или процессы, выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации».

Иерархия уровней безопасности, приведенная в Оранжевой книге, помечает низший уровень безопасности как D, а высший — как A.

В класс D попадают системы, оценка которых выявила их несоответствие требованиям всех других классов.

Основными свойствами, характерными для систем класса C, являются наличие подсистемы учета событий, связанных с безопасностью, и избирательный контроль доступа. Класс (уровень) C делится на два подуровня: уровень C1 обеспечивает защиту данных от ошибок пользователей, но не от действий злоумышленников. На более строгом уровне C2 должны присутствовать:

* средства секретного входа, обеспечивающие идентификацию пользователей путем ввода уникального имени и пароля перед тем, как им будет разрешен доступ к системе;
* избирательный контроль доступа, требуемый на этом уровне, позволяет владельцу ресурса определить, кто имеет доступ к ресурсу и что он может с ним делать (владелец делает это путем предоставления разрешений доступа пользователю или группе пользователей);
* средства аудита (auditing) обеспечивают обнаружение и запись важных событий, связанных с безопасностью, или любых попыток создать системные ресурсы, получить доступ к ним или удалить их;
* защита памяти заключается в том, что память перед ее повторным использованием должна инициализироваться.

На этом уровне система не защищена от ошибок пользователя, но поведение его может быть проконтролировано по записям в журнале, оставленным средствами аудита.

Системы уровня B основаны на помеченных данных и распределении пользователей по категориям, то есть реализуют мандатный контроль доступа. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. Этот уровень в отличие от уровня C защищает систему от ошибочного поведения пользователя.

Уровень A является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня B выполнения формального математически обоснованного доказательства соответствия системы требованиям безопасности.

Различные коммерческие структуры (например, банки) особо выделяют необходимость учетной службы, аналогичной той, что предлагают государственные рекомендации C2. Любая деятельность, связанная с безопасностью, может быть отслежена и тем самым учтена. Это как раз то, что требует стандарт для систем класса C2 и что обычно нужно банкам. Однако коммерческие пользователи, как правило, не хотят расплачиваться производительностью за повышенный уровень безопасности. Уровень безопасности A занимает своими управляющими механизмами до 90 % процессорного времени, что, безусловно, в большинстве случаев неприемлемо. Более безопасные системы не только снижают эффективность, но и существенно ограничивают число доступных прикладных пакетов, которые соответствующим образом могут выполняться в подобной системе. Например, для операционной системы Solaris (версия UNIX) есть несколько тысяч приложений, а для ее аналога уровня B — только около ста.

# Микроядерные операционные системы

В микроядерных операционных системах мы можем выделить центральный компактный модуль, относящийся к супервизорной части системы. Этот модуль имеет очень небольшие размеры и выполняет относительно небольшое количество управляющих функций, но позволяет передать управление на другие управляющие модули, которые и выполнят затребованную функцию. Микроядро — это минимальная главная (стержневая) часть операционной системы, служащая основой модульных и переносимых расширений. Микроядро само является модулем системного программного обеспечения, работающим в наиболее приоритетном состоянии компьютера и поддерживающим связи с остальной частью операционной системы, которая рассматривается как набор серверных приложений (служб).

В 90-е годы XX века было весьма распространенным убеждение, что большинство операционных систем следующих поколений будут строиться как микроядерные. Однако практика показывает, что это не совсем так. Разработчики желают иметь компактное микроядро, но при этом включить в него как можно больше функций, исполняемых непосредственно этим программным модулем. Ибо выполнение затребованной функции другим модулем, вызываемым из микроядра, приводит и к дополнительным задержкам, и к дополнительным сложностям. Более того, имеется масса разных мнений по поводу того, как следует организовывать службы операционной системы по отношению к микроядру; как проектировать драйверы устройств, чтобы добиться наибольшей эффективности, но сохранить функции

драйверов максимально независимыми от аппаратуры; следует ли выполнять операции, не относящиеся к ядру, в пространстве ядра или в пространстве пользователя; стоит ли сохранять программы имеющихся подсистем (например, UNIX) или лучше отбросить все и начать с нуля.

Основная идея, заложенная в технологию микроядра заключается в том, чтобы создать необходимую среду верхнего уровня иерархии, из которой можно легко получить доступ ко всем функциональным возможностям уровня аппаратного обеспечения. При этом микроядро является стартовой точкой для создания всех остальных модулей системы. Все эти остальные модули, реализующие необходимые системе функции, вызываются из микроядра и выполняют сервисную роль. При этом они получают статус обычного процесса или задачи. Можно сказать, что микроядерная архитектура соответствует технологии клиент-сервер. Именно эта технология позволяет в большей мере и с меньшими трудозатратами реализовать перечисленные выше принципы проектирования операционных систем.

Важнейшая задача разработки микроядра заключается в выборе базовых примитивов, которые должны находиться в микроядре для обеспечения необходимого и достаточного сервиса. В микроядре содержится и исполняется минимальное количество кода, необходимое для реализации основных системных вызовов. В число этих вызовов входят передача сообщений и организация другого общения между внешними по отношению к микроядру процессами, поддержка управления прерываниями, а также ряд других весьма немногочисленных функций. Остальные системные функции, характерные для «обычных» (не микроядерных) операционных систем, обеспечиваются как модульные дополнения-процессы, взаимодействующие главным образом между собой и осуществляющие взаимодействие посредством передачи сообщений.

Для большинства микроядерных операционных систем основой для такой архитектуры выступает технология микроядра Mach. Эта операционная система была создана в университете Карнеги Меллон, и многие разработчики брали с нее пример.

Исполняемые микроядром функции ограничены в целях сокращения его размеров и максимизации количества кода, работающего как прикладная программа. Микроядро включает только те функции, которые требуются в целях определения набора абстрактных сред обработки для прикладных программ и организации совместной работы приложений. В результате микроядро обеспечивает только пять различных типов сервисов:

* управление виртуальной памятью;
* поддержка заданий и потоков;
* взаимодействие между процессами (Inter-Process Communication, IPC);
* управление поддержкой ввода-вывода и прерываниями;
* сервисы хоста (host)[1] и процессора.

[1] Хост — главный компьютер. Нынче этим термином обозначают любой компьютер, имеющий IP-адрес.

Другие подсистемы и функции операционной системы, такие как файловые системы, поддержка внешних устройств и традиционные программные интерфейсы, оформляются как системные сервисы либо получают статус обычных обрабатывающих задач. Эти программы работают как приложения на микроядре.

С применением концепции нескольких потоков выполнения на одно задание микроядро создает прикладную среду, обеспечивающую использование мультипроцессоров; при этом совсем не обязательно, чтобы машина была мультипроцессорной: на однопроцессорной машине различные потоки просто выполняются в разное время. Вся поддержка, требуемая для мультипроцессорных машин, сконцентрирована в сравнительно малом и простом микроядре.

Благодаря своим небольшим размерам и способности поддерживать остальные службы в виде обычных процессов, выполняющихся вместе с прикладными программами, сами микроядра проще, чем ядра монолитных или модульных операционных систем. С микроядром супервизорная часть операционной системы разбивается на модульные части, которые могут быть сконфигурированы целым рядом способов, позволяя строить большие системы добавлением частей к меньшим. Например, каждый аппаратно-независимый нейтральный сервис логически отделен и может быть сконфигурирован различными способами. Микроядра также облегчают поддержку мультипроцессоров созданием стандартной программной среды, которая может использовать несколько процессоров, если они есть, однако если их нет, работает на одном. Специализированный код для мультипроцессоров ограничен самим микроядром. Более того, сети из общающихся между собой микроядер могут быть использованы для операционной системной поддержки возникающего класса массивно параллельных машин.

В некоторых случаях использование микроядерного подхода на практике сталкивается с определенными сложностями, что проявляется в некотором замедлении скорости выполнения системных вызовов при передаче сообщений через микроядро по сравнению с классическим подходом. С другой стороны, можно констатировать и обратное. Поскольку микроядра малы и в значительной степени оптимизированы, при соблюдении ряда условий они позволяют обеспечить характеристики реального времени, требующиеся для управления устройствами и для высокоскоростных коммуникаций. Наконец, хорошо структурированные микроядра обеспечивают изолирующий слой для аппаратных различий, которые не маскируются применением языков программирования высокого уровня. Таким образом, они упрощают перенесение кода и увеличивают уровень его повторного использования.

Наиболее ярким представителем микроядерных операционных систем является операционная система реального времени QNX. Микроядро QNX поддерживает только планирование и диспетчеризацию процессов, взаимодействие процессов, обработку прерываний и сетевые службы нижнего уровня (подробнее об ОС QNX см. в главе 10). Это микроядро обеспечивает всего лишь пару десятков системных вызовов, но благодаря этому оно может быть целиком размещено во внутреннем кэше даже таких процессоров, как Intel 486. Как известно, разные версии этой операционной системы имели и разные объемы ядер — от 8 до 46 Кбайт.

Чтобы построить минимальную систему QNX, требуется добавить к микроядру менеджер процессов, который создает процессы и управляет ими и памятью процессов. Чтобы операционная система QNX была применима не только во встроенных и бездисковых системах, нужно добавить файловую систему и менеджер устройств. Эти менеджеры исполняются вне пространства ядра, так что ядро остается небольшим.

# Макроядерные операционные системы

В *макроядерных, или монолитных,* операционных системах ядро, состоящее из множества управляющих модулей и структур данных, не разделено на центральную часть и периферийные (по отношению к этой центральной части) модули. Ядро получается монолитным, неделимым. В этом смысле макроядерные операционные системы являются прямой противоположностью микроядерным. Можно согласиться с тем, как трактуется архитектура монолитных операционных систем в работах [29, 30]. В монолитной операционной системе, несмотря на ее возможную сильную структуризацию, очень трудно удалить один из уровней многоуровневой модульной структуры. Добавление новых функций и изменение существующих для монолитных операционных систем требует очень хорошего знания всей архитектуры операционной системы и чрезвычайно больших усилий.

Очень плодотворным оказался подход, основанный на модели *клиент-сервер.* Эта модель предполагает наличие программного компонента — потребителя какого-либо сервиса, или *клиента,* и программного компонента — поставщика этого сервиса, или *сервера.* Взаимодействие между клиентом и сервером стандартизируется, так что сервер может обслуживать клиентов, реализованных различными способами и, возможно, разными разработчиками. При этом главным требованием является то, чтобы использовался единообразный интерфейс. Инициатором обмена обычно является клиент, который посылает запрос на обслуживание серверу, находящемуся в состоянии ожидания запроса. Один и тот же программный компонент может быть клиентом по отношению к одному виду услуг и сервером для другого вида услуг. Модель клиент-сервер является скорее удобным концептуальным средством ясного представления функций того или иного программного элемента в той или иной ситуации, нежели технологией. Эта модель успешно применяется не только при построении операционных систем, но и на всех уровнях программного обеспечения, и имеет в некоторых случаях более узкий, специфический смысл, сохраняя, естественно, при этом все свои общие черты. Микроядерные операционные системы в полной мере используют модель клиент-сервер.

При поддержке монолитных операционных систем возникает ряд проблем, связанных с тем, что все компоненты макроядра работают в едином адресном пространстве. Во-первых, это опасность возникновения конфликта между различными частями ядра, во-вторых, сложность подключения к ядру новых драйверов. Преимущество микроядерной архитектуры перед макроядерной заключается в том, что каждый компонент системы представляет собой самостоятельный процесс, запуск или остановка которого не отражается на работоспособности остальных процессов.

Микроядерные операционные системы нынче разрабатываются чаще монолитных. Однако следует заметить, что использование технологии клиент-сервер — это еще не гарантия того, что операционная система станет микроядерной. В качестве подтверждения этому можно привести пример с операционными системами класса Windows NT, которые построены на идеологии клиент-сервер, но которые тем не менее трудно назвать микроядерными. Их «микроядро» имеет уже достаточно большой размер, приставка «микро» здесь вызывает улыбку. Хотя по своей архитектуре супервизорная часть этих систем без каких-либо условностей может быть отнесена к системам, построенным на базе модели клиент-сервер. Причем для последних версий операционных систем с общим названием NT (New Technology) еще более заметным является отход от микроядерной архитектуры, но сохранение принципа клиент-сервер во взаимодействиях между модулями управляющей (супервизорной) части. Для того чтобы согласиться с таким высказыванием, достаточно сравнить операционную систему QNX и операционные системы Windows NT/2000/XP.

# Требования к операционным системам реального времени

Как известно, система реального времени (СРВ) должна давать отклик на любые непредсказуемые внешние воздействия в течение предсказуемого интервала времени. Для этого должны выполняться следующие требования.

* *Ограничение времени отклика.* После наступления события реакция на него гарантированно должна последовать до предустановленного крайнего срока. Отсутствие такого ограничения рассматривается как серьезный недостаток программного обеспечения.
* *Одновременность обработки.* Даже если наступает более одного события одновременно, все временные ограничения для всех событий должны быть выдержаны. Это означает, что системе реального времени должен быть присущ параллелизм, что достигается использованием нескольких процессоров и/или многозадачного подхода.

Примерами систем реального времени являются системы управления атомными электростанциями или какими-нибудь технологическими процессами, системы медицинского мониторинга, системы управления вооружением, системы космической навигации, системы разведки, системы управления лабораторными экспериментами, системы управления автомобильными двигателями, робототехника, телеметрические системы управления, системы антиблокировки тормозов, системы сигнализации — список в принципе бесконечен.

Иногда можно услышать из разговоров специалистов, что различают системы «мягкого» и «жесткого» реального времени. Различие между жесткой и мягкой СРВ зависит от требований к системе — система считается жесткой, если «нарушение временных ограничений недопустимо», и мягкой, если «нарушение временных ограничений нежелательно». В недалеком прошлом велось множество дискуссий о точном смысле терминов «жесткая» и «мягкая» СРВ. Можно даже показать, что

мягкая СРВ не является СРВ вовсе, ибо основное требование о соблюдении временных ограничений не выполнено. В действительности термин СРВ часто неправомерно применяют по отношению к быстрым системам.

Часто путают понятия СРВ и ОСРВ (операционная система реального времени), а также неправильно используют атрибуты «мягкая» и «жесткая», когда говорят, что та или иная ОСРВ мягкая или жесткая. Нет мягких или жестких операционных систем реального времени. ОСРВ может только служить основой для построения мягкой или жесткой СРВ. Сама по себе ОСРВ не препятствует тому, что ваша СРВ будет мягкой. Например, пусть вы решили создать СРВ, которая должна работать через Ethernet по протоколу TCP/IP. Такая система не может быть жесткой СРВ, поскольку сама сеть Ethernet в принципе непредсказуема вследствие использования случайного метода доступа к среде передачи данных, в отличие, например, от сети IBM Token Ring или ARC Net, в которых используются детерминированные методы доступа.

Итак, перечислим основные требования к ОСРВ.

## Мультипрограммность и мультизаданность

**Требование 1**. Операционная система должна быть мультипрограммной и мультизадачной *(многопоточной* — multi-threaded), а также активно использовать прерывания для диспетчеризации. Как указывалось выше, ОСРВ должна быть предсказуемой. Это означает не то, что ОСРВ должна быть быстрой, а то, что максимальное время выполнения того или иного действия должно быть известно заранее и соответствовать требованиям приложения. Так, например, система Windows 3.11 даже на Pentium IV с частотой более 3000 МГц не может функционировать в качестве ОСРВ, ибо одно приложение может практически монопольно захватить центральный процессор и заблокировать систему для остальных вычислений.

В соответствии с первым требованием операционная система должна быть многопоточной на принципе абсолютного приоритета (прерываемой). То есть планировщик должен иметь возможность прервать любой поток выполнения и предоставить ресурс тому потоку, которому он более необходим. Операционная система (и аппаратура) должна также обеспечивать прерывания на уровне обработки прерываний.

## Приоритеты задач

**Требование 2.** Должно существовать понятие приоритета потока (задачи). Проблема в том, чтобы определить, какой задаче ресурс требуется более всего. В идеальной ситуации ОСРВ отдает ресурс потоку или драйверу с ближайшим крайним сроком, что называется управлением временным ограничением (DeadLine DrivenOS). Чтобы реализовать это временное ограничение, операционная система должна знать, сколько времени требуется каждому из выполняющихся потоков для завершения. Операционных систем, построенных по этому принципу, практически нет, так как он слишком сложен для реализации. Поэтому разработчики операционных систем принимают иную точку зрения: вводится понятие уровня приоритета для задачи,

и временные ограничения сводятся к приоритетам. Так как умозрительные решения чреваты ошибками, показатели СРВ при этом снижаются. Чтобы более эффективно осуществить указанное преобразование ограничений, проектировщик может воспользоваться теорией расписаний или имитационным моделированием, хотя и это может оказаться бесполезным. Тем не менее, так как на сегодняшний день не имеется иного решения, без понятия приоритета потока не обойтись.

## Наследование приоритетов

**Требование 3.** Должна существовать система наследования приоритетов. На самом деле именно этот механизм синхронизации и тот факт, что различные потоки выполнения используют одно и то же пространство памяти, отличают их от процессов. Как мы уже знаем, процессы не разделяют одно и то же пространство памяти. Так, например, старые версии UNIX не являются многопоточными. «Старая» UNIX — многозадачная ОС, где задачами являются процессы (а не потоки), которые сообщаются через каналы связи (pipes) и разделяемую память. Оба этих механизма используют файловую систему, а ее поведение непредсказуемо.

Комбинация приоритетов потоков и разделение ресурсов между ними приводит к другому явлению — классической проблеме инверсии приоритетов. Это можно проиллюстрировать на примере, где есть как минимум три потока. Когда поток низшего приоритета захватил ресурс, разделяемый с потоком высшего приоритета, и начал выполняться поток среднего приоритета, выполнение потока высшего приоритета будет приостановлено, пока не освободится ресурс и не отработает поток среднего приоритета. В этой ситуации время, необходимое для завершения потока высшего приоритета, зависит от нижних уровней приоритетов — это и есть инверсия приоритетов. Ясно, что в такой ситуации трудно выдержать ограничение на время исполнения.

Чтобы устранить такие инверсии, ОСРВ должна допускать наследование приоритета, то есть повышение уровня приоритета потока до уровня потока, который его вызывает. Наследование означает, что блокирующий ресурс поток наследует приоритет потока, который он блокирует (разумеется, это справедливо лишь в том случае, если блокируемый поток имеет более высокий приоритет).

Иногда можно услышать утверждение, что в грамотно спроектированной системе такая проблема не возникает. В случае сложных систем с этим нельзя согласиться. Единственный способ решения этой проблемы состоит в увеличении приоритета потока «вручную» прежде, чем ресурс окажется заблокированным. Разумеется, это возможно в случае, когда два потока разных приоритетов претендуют на один ресурс. В общем случае решения не существует.

## Сихронизация процессов и задач

**Требование** 4. Операционная система должна обеспечивать мощные, надежные **и** удобные механизмы синхронизации задач. Так как задачи разделяют данные (ресурсы) **и** должны сообщаться друг с другом, представляется логичным существование механизмов блокирования **и** коммуникации. То есть необходимы механиз-

мы, гарантированно предоставляющие возможность оперативно обменяться сообщениями и синхросигналами между параллельно выполняющимися задачами и процессами. Эти системные механизмы должны быть всегда доступны процессам, требующим реального времени. Следовательно, системные ресурсы для их функционирования должны быть распределены заранее.

## Предсказуемость

**Требование** 5. Поведение операционной системы должно быть известно **и** достаточно точно прогнозируемо. Времена выполнения системных вызовов **и** временные характеристики поведения системы в различных обстоятельствах должны быть известны разработчику. Поэтому создатель ОСРВ должен приводить следующие характеристики:

* латентную задержку прерывания, то есть время от момента прерывания до момента запуска задачи: она должна быть предсказуема **и** согласована с требованиями приложения (эта величина зависит от числа одновременно «висящих» прерываний);
* максимальное время выполнения каждого системного вызова (оно должно быть предсказуемо **и не** должно зависеть от числа объектов в системе);
* максимальное время маскирования прерываний драйверами **и** супервизорными модулями операционной системы.

# Интерфейсы операционных систем

Напомним, что операционная система всегда выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами. Под интерфейсами операционных систем здесь и далее следует понимать специальные интерфейсы системного и прикладного программирования (API), предназначенные для выполнения перечисленных ниже задач.

* Управление процессами, которое включает в себя следующий набор основных функций:
  - запуск, приостанов и снятие задачи с выполнения;
  - задание или изменение приоритета задачи;
  - взаимодействие задач между собой (механизмы сигналов, семафорные примитивы, очереди, конвейеры, почтовые ящики);
  - вызов удаленных процедур (Remote Procedure Call, RPC).
* Управление памятью:
  - запрос на выделение блока памяти;
  - освобождение памяти;
  - изменение параметров блока памяти (например, память может быть заблокирована процессом либо предоставлена в общий доступ);
  - отображение файлов на память (имеется не во всех системах).

* Управление вводом-выводом:
  - запрос на управление виртуальными устройствами (напомним, что управление вводом-выводом является привилегированной функцией самой операционной системы, и никакая из пользовательских задач не должна иметь возможности непосредственно управлять устройствами);
  - файловые операции (запросы к системе управления файлами на создание, изменение и удаление данных, организованных в файлы).

Здесь мы перечислили основные наборы функций, которые выполняются операционной системой по соответствующим запросам от задач. Что касается интерфейса пользователя с операционной системой, то он реализуется с помощью специальных программных модулей, которые принимают его команды на соответствующем языке (возможно, с использованием графического интерфейса) и транслируют их в обычные вызовы в соответствии с основным интерфейсом системы. Обычно эти модули называют интерпретатором команд. Так, например, функции такого интерпретатора в MS DOS выполняет модуль COMMAND.COM. Получив от пользователя команду, такой модуль после лексического и синтаксического анализа либо сам выполняет действие, либо, что случается чаще, обращается к другим модулям операционной системы, используя механизм API. Надо заметить, что в последние годы большую популярность получили *графические интерфейсы* (Graphical User Interface, GUI), в которых задействованы соответствующие манипуляторы типа мышь или трекбол (track-ball)[1]. Указание курсором на объект и щелчок или двойной щелчок на соответствующей кнопке мыши приводит к каким-либо действиям — запуску программы, ассоциированной с объектом, выбору и/или активизации меню и т. д. Можно сказать, что такая интерфейсная подсистема транслирует «команды» пользователя в обращения к операционной системе.

Поясним также, что управление GUI является частным случаем задачи управления вводом-выводом и не относится к функциям ядра операционной системы, хотя в ряде случаев разработчики операционной системы относят функции GUI к основному системному интерфейсу API.

Следует отметить, что имеются два основных подхода к управлению задачами. Так, в одних системах порождаемая задача наследует все ресурсы задачи-родителя, тогда как в других системах существуют равноправные отношения, и при порождении нового процесса ресурсы для него запрашиваются у операционной системы.

Обращения к операционной системе в соответствии с имеющимся интерфейсом API могут осуществляться как посредством вызова подпрограммы с передачей ей необходимых параметров, так и через механизм программных прерываний. Выбор метода реализации вызовов функций API должен определяться архитектурой платформы.

[1] Трекбол — специальный шарик, который в переносных компьютерах (NoteBook) размещается рядом с клавиатурой, прокручивается пальцами и служит для перемещения указателя мыши. В настоящее время гораздо чаще используют устройство, чувствительное к касанию (touchpad). С помощью такого устройства пользователь управляет указателем мыши, перемещая палец по специальной поверхности.

Так, например, в операционной системе MS DOS, которая разрабатывалась для однозадачного режима (поскольку процессор i80x86 не поддерживал мультипрограммирование), использовался механизм программных прерываний. При этом основной набор функций API был доступен через точку входа обработчика int 21 h.

В более сложных системах имеется не одна точка входа, а множество — по количеству функций API. Так, в большинстве операционных систем используется метод вызова подпрограмм. В этом случае вызов сначала передается в модуль API, например в библиотеку времени выполнения (Run Time Library, RTL)[1], который перенаправляет его соответствующим обработчикам программных прерываний, входящим в состав операционной системы. Использование механизма прерываний вызвано, главным образом, тем, что при этом процессор переводится в режим супервизора.

# Интерфейс прикладного программирования

Прежде всего, необходимо однозначно разделить общий термин *API* (Application Program Interface — интерфейс прикладного программирования) на следующие направления:

* API как интерфейс высокого уровня, принадлежащий к библиотекам RTL;
* API прикладных и системных программ, входящих в поставку операционной системы;
* прочие интерфейсы API.

*Интерфейс прикладного программирования,* как это и следует из названия, предназначен для использования прикладными программами системных ресурсов компьютера и реализуемых операционной системой разнообразных системных функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам операционной системы.

Итак, API — это набор функций, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной программы с целевой вычислительной системой. Целевая вычислительная система представляет собой совокупность программных и аппаратных средств, в окружении которых выполняется результирующая программа. Сама результирующая программа порождается системой программирования на основании кода исходной программы, созданного разработчиком, а также объектных модулей и библиотек, входящих в состав системы программирования.

В принципе API используется не только прикладными, но и системными программами как в составе операционной системы, так и в составе системы программирования. Но дальше речь пойдет только о функциях API с точки рения разработчика

[1] RTL включает в себя те стандартные подпрограммы, которые система программирования подставляет на этапе компиляции. В общем случае это не только модули системы программирования, но и модули самой операционной системы.

прикладной программы. Для системной программы существуют некоторые дополнительные ограничения на возможные реализации API.

Функции API позволяют разработчику строить результирующую прикладную программу так, чтобы использовать средства целевой вычислительной системы для выполнения типовых операций. При этом разработчик программы избавлен от необходимости создавать исходный код для выполнения этих операций.

Программный интерфейс API включает в себя не только сами функции, но и соглашения об их использовании, которые регламентируются операционной системой, архитектурой целевой вычислительной системы и системой программирования.

Существует несколько вариантов реализации API:

* реализация на уровне модулей операционной системы;
* реализация на уровне системы программирования;
* реализация на уровне внешней библиотеки процедур и функций.

Система программирования в каждом из этих вариантов предоставляет разработчику средства для подключения функций API к исходному коду программы и организации их вызовов. Объектный код функций API подключается к результирующей программе компоновщиком при необходимости.

Возможности API можно оценивать со следующих позиций:

* эффективности выполнения функций API — эффективность включает в себя скорость выполнения функций и объем вычислительных ресурсов, необходимых для их выполнения;
* широты предоставляемых возможностей;
* зависимости прикладной программы от архитектуры целевой вычислительной системы.

В идеале хотелось бы иметь набор функций API, выполняющихся с наивысшей эффективностью, предоставляющих пользователю все возможности современных операционных систем и имеющих минимальную зависимость от архитектуры вычислительной системы (еще лучше — лишенных такой зависимости).

Добиться наивысшей эффективности выполнения функций API практически трудно по тем же причинам, по которым невозможно добиться наивысшей эффективности выполнения для любой результирующей программы. Поэтому об эффективности интерфейса API можно говорить только в сравнении его характеристик с другими интерфейсами API.

Что касается двух других показателей, то в принципе нет никаких технических ограничений на их реализацию. Однако существуют организационные проблемы и узкие корпоративные интересы, тормозящие создание такого рода библиотек.

## Реализация функций API на уровне модулей операционной системы

При реализации функций API на уровне модулей операционной системы операционная система ответственна за выполнение функций API. Объектный код, вы-

полняющий функции, либо непосредственно входит в состав операционной системы (или даже ядра операционной системы), либо находится в составе динамически загружаемых библиотек, поставляемых вместе с системой. Система программирования ответственна только за то, чтобы организовать интерфейс для вызова этого кода.

В таком варианте результирующая программа обращается непосредственно к операционной системе. Поэтому достигается наибольшая эффективность выполнения функций API по сравнению со всеми другими вариантами реализации API.

Недостатком организации API по такой схеме является практически полное отсутствие переносимости не только кода результирующей программы, но и кода исходной программы. Программа, созданная для одной архитектуры вычислительной системы, не сможет исполняться на вычислительной системе другой архитектуры даже после того, как ее объектный код полностью перестроен. Чаще всего система программирования просто не сможет выполнить перестроение исходного кода для новой архитектуры вычислительной системы, поскольку многие функции API, ориентированные на определенную операционную систему, в новой архитектуре могут просто отсутствовать.

Таким образом, в данной схеме для переноса прикладной программы с одной целевой вычислительной системы на другую потребуется изменение исходного кода программы.

Переносимости можно было бы добиться, если унифицировать функции API в различных операционных системах. С учетом корпоративных интересов производителей операционных систем и иного системного программного обеспечения такое направление их развития представляется практически невозможным. В лучшем случае при приложении определенных организационных усилий удается добиться стандартизации смыслового (семантического) наполнения основных функций API, но не их программного интерфейса.

Хорошо известным примером API такого рода может служить набор функций, предоставляемых пользователю со стороны операционных систем типа Microsoft Windows — WinAPI (Windows API). Надо сказать, что даже внутри этого корпоративного интерфейса API существует определенная несогласованность, которая несколько ограничивает переносимость программ между различными операционными системами типа Windows. Еще одним примером такого интерфейса API можно считать набор сервисных функций простейших однопрограммных операционных систем типа MS DOS, реализованный в виде набора подпрограмм обслуживания программных прерываний.

## Реализация функций API на уровне системы программирования

При реализации функций API на уровне системы программирования эти функции предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Обычно речь идет о библиотеке времени выполнения (RTL). Система программирования предоставляет пользователю библиотеку

функций и обеспечивает подключение к результирующей программе объектного кода, ответственного за выполнение этих функций.

Очевидно, что эффективность вызова функций API в таком варианте будет несколько ниже, чем при непосредственном обращении к функциям операционной системы. Так происходит, поскольку для выполнения многих функций API библиотека RTL языка программирования должна все равно выполнять обращения к функциям операционной системы. Наличие всех необходимых вызовов и обращений к функциям операционной системы в объектном коде RTL обеспечивает система программирования.

Однако переносимость исходного кода программы в таком варианте оказывается самой высокой, поскольку синтаксис и семантика всех функций строго регламентированы в стандарте соответствующего языка программирования. Они зависят от языка и не зависят от архитектуры целевой вычислительной системы. Поэтому для выполнения прикладной программы на новой архитектуре вычислительной системы достаточно заново построить код результирующей программы с помощью соответствующей системы программирования.

Единообразное выполнение функций языка обеспечивается системой программирования. При ориентации на различные архитектуры целевой вычислительной системы в системе программирования могут потребоваться различные комбинации вызовов функций операционной системы для выполнения одних и тех же функций исходного языка. Это должно быть учтено в коде RTL. В общем случае для каждой архитектуры целевой вычислительной системы потребуется свой код RTL языка программирования. Выбор того или иного объектного кода RTL для подключения к результирующей программе система программирования обеспечивает автоматически.

Например, рассмотрим функции динамического выделения памяти в языках C и Паскаль. В C это функции malloc, realloc и free (функции new и delete в C++), в Паскале — функции new и dispose. Если использовать эти функции в исходном тексте программы, то с точки зрения исходной программы они будут действовать одинаковым образом в зависимости только от семантики исходного кода программы. При этом для разработчика исходной программы не имеет значения, на какую архитектуру ориентирована его программа. Это имеет значение для системы программирования, которая для каждой из этих функций должна подключить к результирующей программе объектный код библиотеки. Этот код будет выполнять обращение к соответствующим функциям операционной системы. Не исключено даже, что для однотипных по смыслу функций в разных языках (например, malloc в C и new в Паскале выполняют схожие по смыслу действия) этот код будет выполнять схожие обращения к операционной системе. Однако для различных вариантов операционной системы этот код будет различен даже при использовании одного и того же исходного языка.

Проблема главным образом заключается в том, что большинство языков программирования предоставляют пользователю не очень широкий набор стандартизованных функций. Поэтому разработчик исходного кода существенно ограничен в выборе доступных функций API. Как правило, набора стандартных функций ока-

зывается недостаточно для создания полноценной прикладной программы. Тогда разработчик может обратиться к функциям других библиотек, имеющихся в составе системы программирования. В этом случае нет гарантии, что функции, включенные в состав данной системы программирования, но не входящие в стандарт языка программирования, будут доступны в другой системе программирования, особенно если та ориентирована на другую архитектуру целевой вычислительной системы. Такая ситуация уже ближе к третьему варианту реализации API.

Например, те же функции malloc, realloc и free в языке C фактически не входят в стандарт языка. Они входят в состав стандартной библиотеки, которая «де-факто» входит во все системы программирования, построенные на основе языка C. Общепринятые стандарты существуют для многих часто используемых функций языка. Если же взять более специфические функции, такие как функции порождения новых процессов, то для них ни в C, ни в Паскале не окажется общепринятого стандарта.

## Реализация функций API с помощью внешних библиотек

При реализации функций API с помощью внешних библиотек эти функции предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком.

Система программирования ответственна только за то, чтобы подключить объектный код библиотеки к результирующей программе. Причем внешняя библиотека может быть и динамически загружаемой (загружаемой во время выполнения программы).

С точки зрения эффективности выполнения этот метод реализации API имеет самые низкие результаты, поскольку внешняя библиотека обращается как к функциям операционной системы, так и к функциям RTL языка программирования. Только при очень высоком качестве внешней библиотеки ее эффективность сравнима *с* эффективностью библиотеки RTL

Если говорить о переносимости исходного кода, то здесь требование только одно — используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована прикладная программа. Тогда удается достигнуть переносимости. Это возможно, если внешняя библиотека удовлетворяет какому-то принятому стандарту, а система программирования поддерживает этот стандарт.

Например, библиотеки, удовлетворяющие стандарту POSIX (см. следующий раздел), доступны в большинстве систем программирования для языка C. И если прикладная программа использует только библиотеки этого стандарта, то ее исходный код будет переносимым. Еще одним примером является широко известная библиотека графического интерфейса XLib, поддерживающая стандарт графической среды X-Window.

Для большинства специфических библиотек отдельных разработчиков это не так. Если пользователь использует какую-то библиотеку, то она ориентирована на ог-

раниченный набор доступных архитектур целевой вычислительной системы. Примерами могут служить библиотеки MFC (Microsoft Foundation Classes) от Microsoft и VCL (Visual Controls Library) от Borland, жестко ориентированные на архитектуру операционных систем семейства Windows.

Тем не менее многие фирмы-разработчики сейчас стремятся создавать библиотеки, которые бы обеспечивали переносимость исходного кода приложений между различными архитектурами целевой вычислительной системы. Пока еще такие библиотеки не получили широкого распространения, но имеется несколько попыток их реализации, например библиотека CLX от Borland ориентирована на архитектуру операционных систем семейств Linux и Windows.

В целом развитие функций API идет в направлении попытки создать библиотеки API, обеспечивающие широкую переносимость исходного кода. Однако с учетом корпоративных интересов различных производителей и сложившейся ситуации на рынке в ближайшее время вряд ли удастся достичь значительных успехов в этом направлении. Разработка широко применимого стандарта API пока еще остается делом будущего.

Поэтому разработчики системных программ вынуждены оставаться в довольно узких рамках ограничений стандартных библиотек языков программирования.

Что касается прикладных программ, то гораздо большую перспективу для них предоставляют технологии, связанные с разработками в рамках архитектуры клиент-сервер или трехзвенной архитектуры создания приложений. В этом направлении ведущие производители операционных систем, СУБД и систем программирования скорее достигнут соглашений, чем в направлении стандартизации API.

Итак, нами были рассмотрены основные принципы, цели и подходы к реализации системных интерфейсов API. Отметим еще один очень важный момент: желательно, чтобы интерфейс прикладного программирования не зависел от системы программирования. Конечно, были одно время персональные компьютеры, у которых базовой системой программирования выступал интерпретатор с языка Basic, но это скорее исключение. Обычно интерфейс API не зависит от системы программирования и может вызываться из любой системы программирования, хотя и с использованием соответствующих правил построения вызывающих последовательностей. В то же время, в ряде случаев система программирования может сама генерировать обращения к API. Например, мы можем написать в программе вызов функции по запросу 256 байт памяти:

```
unsigned char * ptr = malloc (256).
```

Система программирования с языка C сгенерирует целую последовательность обращений. Из кода пользовательской программы будет осуществлен вызов библиотечной функции malloc, код которой расположен в RTL языка C. Библиотека времени выполнения, в данном случае, реализует вызов malloc уже как вызов системной функции HeapAlloc API:

```
LPVOID HeapAlloc(
    HANDLE  hHeap,  // handle to the private heap block - указатель на блок
    DWORD  dwFlags, // heap allocation control flags - свойства блока
```

```
    DWORD  dwBytes  // number of bytes to allocate - размер блока
);
```

Параметры выделяемого блока памяти в таком случае задаются системой программирования, и пользователь лишен возможности задавать их напрямую. С другой стороны, если это необходимо, функции API можно вызывать прямо в тексте программы:

```
unsigned char * ptr = (LPVOID) HeapAlloc( GetProcessHeap(), 0, 256);
```

В этом случае программирование вызова немного усложняется, но получаемый конечный результат будет, как правило, короче и, что самое важное, работать будет эффективнее. Следует отметить, что далеко не все возможности API доступны через обращения к функциям системы программирования. Непосредственное обращение к API позволяет пользователю обращаться к системным ресурсам более эффективным способом. Однако это требует знания функций API, количество которых нередко достигает нескольких сотен.

Как правило, функции API не стандартизированы. В каждом конкретном случае набор вызовов API определяется, прежде всего, архитектурой операционной системы и ее назначением. В то же время, принимаются попытки стандартизировать некоторый базовый набор функций, поскольку это существенно облегчило бы перенос приложений с одной операционной системы на другую. Таким примером может служить очень известный и, пожалуй, один из самых распространенных стандарт POSIX. В этом стандарте перечислен большой набор функций, их параметров и возвращаемых значений. Стандартизированными, согласно POSIX, являются не только обращения к API, но и файловая система, организация доступа к внешним устройствам, набор системных команд[1]. Использование в приложениях этого стандарта позволяет в дальнейшем легко переносить такие программы с одной операционной системы в другую путем простейшей перекомпиляции исходного текста.

Частным случаем попытки стандартизации API является внутренний корпоративный стандарт компании Microsoft, известный как WinAPI. Он включает в себя следующие реализации: Win16, Win32s, Win32, WinCE. С точки зрения WinAPI (в силу ряда идеологических причин графический, то есть «оконный», интерфейс пользователя обязателен) базовой задачей является окно. Таким образом, стандарт WinAPI изначально ориентирован на работу в графической среде. Однако базовые понятия дополнены традиционными функциями, в том числе частично поддерживается стандарт POSIX.

# Интерфейс POSIX

POSIX (Portable Operating System Interface for Computer Environments — независимый от платформы системный интерфейс для компьютерного окружения) — это стандарт IEEE (Institute of Electrical and Electronics Engineers — институт инженеров по электротехнике и радиоэлектронике), описывающий системные интер-

[1] В данном контексте под системными командами следует понимать некий набор программ, позволяющих управлять вычислительными процессами, например pstat, kill, dir и др.

фейсы для открытых операционных систем, в том числе оболочки, утилиты и инструментарии. Помимо этого, согласно POSIX, стандартизированными являются задачи обеспечения безопасности, задачи реального времени, процессы администрирования, сетевые функции и обработка транзакций. Стандарт базируется на UNIX-системах, но допускает реализацию и в других операционных системах.

Интерфейс POSIX начинался как попытка пропаганды институтом IEEE идей переносимости приложений в UNIX-средах путем разработки абстрактного независимого от платформы стандарта. Однако POSIX не ограничивается только UNIX-системами; существуют различные реализации этого стандарта в системах, которые соответствуют требованиям, предъявляемым стандартом IEEE Standard 1003.1-1990 (POSIX.1). Например, известная ОС реального времени QNX соответствует спецификациям этого стандарта, что облегчает перенос приложений в эту систему, но UNIX-системой не является ни в каком виде, ибо ее архитектура использует абсолютно иные принципы.

Этот стандарт подробно описывает систему виртуальной памяти (Virtual Memory System, VMS), многозадачность (Multiprocess Executing, MPE) и технологию переноса операционных систем (CTOS). Таким образом, на самом деле POSIX представляет собой множество стандартов POSIX.1-POSIX.12. В табл. 9.1 перечислены основные направления, описываемые данными стандартами. Следует также особо отметить, что в POSIX.1 основным языком описания системных функций API предполагается язык C.

**Таблица 9.1.** Семейство стандартов POSIX

| Стандарт | Стандарт ISO | Краткое описание |
|---|---|---|
| POSIX.0 | Нет | Введение в стандарт открытых систем. Данный документ не является стандартом в чистом виде, а представляет собой рекомендации и краткий обзор технологий |
| POSIX.1 | Да | Системный интерфейс API (язык C) |
| POSIX.2 | Нет | Оболочки и утилиты (одобренные IEEE) |
| POSIX.3 | Нет | Тестирование и верификация |
| POSIX.4 | Нет | Задачи реального времени и потоки выполнения |
| POSIX.5 | Да | Использование языка ADA применительно к стандарту POSIX.1 |
| POSIX.6 | Нет | Системная безопасность |
| POSIX.7 | Нет | Администрирование системы |
| POSIX.8 | Нет | Сети, «прозрачный» доступ к файлам, абстрактные сетевые интерфейсы, не зависящие от физических протоколов, вызовы RPC, связь системы с приложениями, зависящими от протокола |
| POSIX.9 | Да | Использование языка Fortran, применительно к стандарту POSIX.1 |
| POSIX.10 | Нет | Super-computing Application Environment Profile (AEP) |
| POSIX.11 | Нет | Обработка транзакций AEP |
| POSIX.12 | Нет | Графический интерфейс пользователя (GUI) |

Таким образом, программы, написанные с соблюдением данных стандартов, будут одинаково выполняться на всех POSIX-совместимых системах. Однако стандарты отчасти носят всего лишь рекомендательный характер. Часть стандартов описана очень строго, тогда как другая часть только поверхностно раскрывает основные требования. Нередко программные системы заявляются как POSIX-совместимые, хотя таковыми их назвать нельзя. Причины кроются в формальном подходе к реализации интерфейса POSIX в различных операционных системах. На рис. 9.1 изображена типовая схема реализации строго соответствующего POSIX приложения.

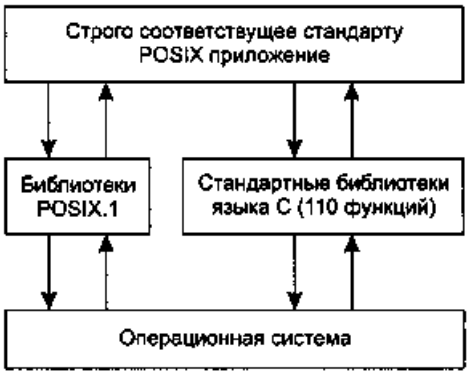


**Рис.** 9.1. Схема реализации приложения, строго соответствующего стандарту POSIX

Из рисунка видно, что для взаимодействия с операционной системой программа использует только библиотеки POSIX.1 и стандартную библиотеку RTL языка С, в которой возможно использование только 110 различных функций, также описанных стандартом POSIX.1.

К сожалению, достаточно часто с целью увеличения производительности той или иной подсистемы либо для введения фирменных технологий, которые ограничивают область применения приложения соответствующей операционной средой, при программировании используются другие функции, не отвечающие стандарту POSIX.

Реализации стандарта POSIX на уровне операционной системы различны. Если UNIX-системы в своем абсолютном большинстве изначально соответствуют спецификациям IEEE Standard 1003.1-1990, то WinAPI не является POSIX-совместимым. Однако для его поддержки в операционной системе Windows NT введен специальный модуль API для поддержки стандарта POSIX, работающий на уровне привилегий пользовательских процессов. Данный модуль обеспечивает преобразование и передачу вызовов из пользовательской программы к ядру системы и обратно, работая с ядром через WinAPI. Прочие приложения, написанные с использованием WinAPI, могут передавать информацию POSIX приложениям через стандартные механизмы потоков ввода-вывода stdin и stdout [37].

# Примеры программирования для разных интерфейсов API

Для наглядной демонстрации принципиальных различий интерфейсов API наиболее популярных современных операционных систем для персональных компьютеров рассмотрим простейший пример, в котором необходимо подсчитать количество пробелов в текстовых файлах, имена которых должны указываться в командной строке. Рассмотрим два варианта программы: для Windows (с использованием WinAPI) и для Linux (POSIX API).

Поскольку нас интересует работа с параллельными задачами, пусть при выполнении программы для каждого из перечисленных в командной строке файлов создается свой процесс или поток выполнения (задача), который параллельно с другими процессами (потоками) производит работу по подсчету пробелов в «своем» файле. Результатом работы программы будет являться список файлов с подсчитанным количеством пробелов для каждого.

Следует обратить особое внимание на то, что приведенные ниже реализации программ решения данной задачи не являются единственно возможными. В обеих рассматриваемых операционных системах существуют разные методы работы с файловой системой и управления процессами. В данном случае рассматривается только один, но наиболее характерный для соответствующего интерфейса API вариант.

Для того чтобы было удобнее сравнивать эту (листинг 9.1) и следующую (листинг 9.2) программы, а также учитывая, что задача не требует для своего решения оконного интерфейса, в тексте использованы только те вызовы API, которые не затрагивают графический интерфейс. Конечно, нынче редко какое приложение не использует возможностей GUI, но зато в нашем случае сразу можно увидеть разницу в организации параллельной работы запускаемых вычислений.

**Листинг 9.1.** Текст программы для Windows (WinAPI)

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

// Название processFile
// Описание исполняемый код потока
// Входные параметры lpFileName - имя файла для обработки
// Выходные параметры нет
DWORD processFile(LPVOID lpFileName ) {
    HANDLE handle, // описатель файла
    DWORD numRead, total = 0,
    char buf,

    // запрос к ОС на открытие файла (только для чтения)
    handle = CreateFile( (LPCTSTR)lpFileName GENERIC_READ.
      FILE_SHARE_READ, NULL, OPEN_EXISTING FILE_ATTRIBUTE_NORMAL, NULL);

    // цикл чтения до конца файла
    do {
```

*продолжение &*

**Листинг 9.1** (*продолжение*)

```
            // чтение одного символа из файла
            ReadFile( handle, (LPVOID) &buf, 1, &numRead, NULL);
            if (buf == 0x20) total++;
    } while ( numRead > 0);

    fprintf( stderr, "(ThreadID: %Lu), File %s, spaces = %d\n",
      GetCurrentThreadId(), lpFileName, total);

    // закрытие файла
    CloseHandle( handle);

    return(0);
}

// Название: main
// Описание: главная программа
// Входные параметры: список имен файлов для обработки
// Выходные параметры: нет
int main(int argc, char *argv[]) {
int i;
DWORD pid;
HANDLE hThrd[255];    // массив ссылок на потоки

    // для всех файлов, перечисленных в командной строке
    for (i = 0; i< (argc-1); i++) {
        // запуск потока - обработка одного файла
        hThrd[i] = CreateThread( NULL, 0x4000,
          (LPTHREAD_START_ROUTINE) processFile,
          (LPVOID) argv[i+1], 0, &pid);
        fprintf( stdout, "processFile started (HND=%d)\n", hThrd[i]);
    }

    // ожидание окончания выполнения всех запущенных потоков
    WaitForMultipleObjects( argc-1, hThrd, true, INFINITE);

    return(0);
}
```

Обратите внимание, что основная программа запускает потоки и ждет окончания их выполнения. Другими словами, мы имеем всего один вычислительный процесс, но используем мультизадачные возможности операционной системы.

**Листинг 9.2.** Текст программы для Linux (POSIX API)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <wait.h>
#include <fcntl.h>
#include <stdio.h>

// Название: processFile
// Описание: обработка файла, подсчет кол-ва пробелов
// Входные параметры: fileName - имя файла для обработки
// Выходные параметры: кол-во пробелов в файле
int processFile( char *fileName) {
    int handle, numRead, total = 0;
```

```
    char buf;

    // запрос к ОС на открытие файла (только для чтения)
    handle = open( fileName, O_RDONLY);

    // цикл чтения до конца файла
    do {
        // чтение одного символа из файла
        numRead = read( handle, &buf, 1);
        if (buf == 0x20) total++;
    } while (numRead > 0);

       // закрытие файла
    close( handle);
    return( total);
}

// Название main
// Описание главная программа
// Входные параметры: список имен файлов для обработки
// Выходные параметры нет
int main(int argc, char *argv[]) {
int i, pid, status;

    // для всех файлов, перечисленных в командной строке
    for (i = 1; i< argc; i++) {
        // запускаем дочерний процесс
        pid = fork();
        if (pid == 0) {
            // если выполняется дочерний процесс
            // вызов функции счета количества пробелов в файле
            printf( "(PID. %d), File %s, spaces = %d\n",
              getpid(), argv[i], processFile( argv[i]));
            // выход из процесса
            exit();
        }
        // если выполняется родительский процесс
        else
        printf( "processFile started (pid=%d)\n", pid);
    }

    // ожидание окончания выполнения всех запущенных процессов
    if (pid != 0) while (wait(&status)>0);
    return;
}
```

Из листинга 9.2 видно, что здесь все вычисления имеют статус процессов, а не потоков выполнения. Надо заметить, что многие современные версии UNIX поддерживают механизм потоков, поскольку потоки в ряде случаев позволяют повысить эффективность вычислений и упрощают их создание, но в рассматриваемом интерфейсе потоков нет.

В заключение можно заметить, что очень трудно сравнивать интерфейсы API. При их разработке создатели, как правило, стараются реализовать функционально полный набор основных функций, используя которые можно решать разные задачи, правда, порой, различными способами. Один набор системных функций хорош для

одного набора задач, другой — для иного набора задач. Тем более что, фактически, сейчас мы имеем существенно ограниченное множество интерфейсов API из-за того, что имеет место доминирование наиболее популярных операционных систем и на их распространении в большей степени сказалась правильная маркетинговая политика их создателей, а не достоинства и недостатки самих этих систем и их интерфейсов.

## Контрольные вопросы и задачи

1. Что вы понимаете под архитектурой операционной системы?
2. Перечислите и поясните основные принципы построения операционных систем.
3. Для чего операционные системы используют несколько режимов работы процессора? Чем отличается супервизорный режим работы процессора от пользовательского? Как часто процессор переводится в супервизорный режим?
4. Объясните принцип виртуализации. Имеется ли связь между принципом виртуализации и принципом совместимости? Если имеется, то поясните, в чем она заключается?
5. Что такое ядро операционной системы? Расскажите об основных моментах, характерных для микроядерных ОС. Какие основные функции должно выполнять микроядро ОС?
6. Перечислите основные требования, предъявляемые к операционным системам в плане обеспечения информационной безопасности.
7. Перечислите основные требования, предъявляемые к операционным системам реального времени.
8. Какие задачи возлагаются на интерфейс прикладного программирования (API)?
9. Какими могут быть варианты реализации API? В чем заключаются достоинства и недостатки каждого варианта?
10. Что такое библиотека времени выполнения (RTL)?
11. Что такое POSIX? Какими преимуществами обладают программы, созданные с использованием только стандартных функций, предусмотренных POSIX?